Functional Programming and Quadtrees[1]

F. Warren Burton
School of Computing Sciences
Simon Fraser University
Burnaby
British Columbia
Canada V5A 1S6
Email: burton@cs.utah.edu

John (Yannis) G. Kollias
Department of Electrical Engineering
Division of Computer Science
National Technical University of Athens
157 73 Zographou, Athens
Greece
Email: mcvax!ariadne!theseas!kollias

March 4, 2008

[1]This material is based upon work supported by the National Science Foundation under Grant No. MIP-8796167. Travel support was provided by NATO under collaborative Research Grant No. 401/84.

Abstract

Modern functional programming languages offer many advantages to the programmer of graphics algorithms.

A functional program is a collection of recursive mathematical equations. Functional programs are usually much easier than procedural programs to read, write, verify and modify. A good modern functional programming language will support various kinds of user defined data types, polymorphic type checking, good facilities for modularity, pattern matching, and similar modern language features commonly found in other types of languages. In addition, a modern functional programming language should support lazy evaluation and higher order functions.

With lazy evaluation, nothing is evaluated until its value is required for the final output. This leads to time efficiencies, because often large parts of intermediate results in graphics problems are later discarded. These discarded parts are never generated with lazy evaluation. Lazy evaluation also leads to space efficiencies, because graphical objects can be generated piecewise, with the pieces reclaimed by the storage management system after they have been used.

Higher order functions are functions that take other functions as arguments and/or return functions as results. In a functional language, some higher order functions play a role similar to control structures in procedural languages. A programmer can write his own higher order functions to define new "control structures" suitable for the data structures and applications in which he is interested.

We will illustrate some of the advantages of functional programming languages, using the language Miranda, in the context of quadtree algorithms.

1 Introduction

Modern functional programming languages offer many advantages to the programmer of graphics algorithms. We will illustrate some of these in the context of quadtree algorithms.

A functional program is a collection of recursive mathematical equations. Functional programming languages and logic programming languages are often called declarative languages, since a program can be considered a static mathematical declaration of what is required. However, programs in both kinds of languages can be viewed operationally. That is, a program specifies an algorithm as well as a result. We will be primarily interested in the operational view of functional programs.

A good modern functional programming language will support various kinds of user defined data types, polymorphic type checking, good facilities for modularity, pattern matching, and similar modern language features commonly found in other types of languages. Functional programming languages do not support assignment statements or other state changing statements, but do have some important features not found in most conventional languages. These include lazy evaluation and higher order functions.

A lazy functional language never evaluates an expression until it has been determined that the value of the expression is required for the end result of the computation. This is particularly useful in graphics and computational geometry, where it is often useful to have intermediate results that are largely discarded later (because much of the result is hidden or fails to intersect another object, etc.) These discarded portions are never constructed with lazy evaluation.

Another advantage of lazy evaluation in a functional language is that the evaluation of an object is delayed until that object is needed, and then it is evaluated piecewise as it is required. This produces a coroutine like behavior. In graphics or computational geometry, where one often works with unordered spatial objects, it is useful to have a declarative language where the order in which objects are declared is not necessarily the order in which they are computed.

Higher order functions are functions that take other functions as arguments and/or return functions as results. In a functional language, some higher order functions play a role similar to control structures in procedural languages. A programmer can write his own higher order functions to define new "control structures" suitable for the data structures and applications in which he is interested. Some procedural languages, including Algol 68 and some versions of Lisp, support higher order functions. Lazy evaluation can be simulated using higher order functions.

Perhaps the greatest advantage of a functional program over a procedural program (written in a language such as Pascal, Fortran, C or Ada) is that a functional program tends to be much simpler, and hence to be easier to write, understand and modify. Since a functional program is a collection of mathematical equations, functional programs can be manipulated using the ordinary laws of mathematics. This makes it easier to formally derive, transform or verify a functional program. Because of this, functional programs are often used as executable program specifications in situations where the final product must be implemented in a nonfunctional language. The reader may choose to view a functional program as either an executable formal specification of an algorithm, or as a program in its own right.

Some recent functional language implementations are said to rival the speed of conventional languages [**?**, p. xvii]. However, the best procedural language implementations, which do not need to support higher

order functions, are likely to always perform better than the best functional language implementations.

The examples in this paper are written in the Miranda[1] functional programming language [?]. We have choosen to use Miranda since it is the most widely used modern functional programming language. As of February, 1988, Miranda was in use at 200 sites, including 35 industrial sites. The current implementation of Miranda is very slow, making it unsuitable for production programming in applications where efficiency is critical. A number of other functional languages have the features in which we are interested. The techniques discussed in this paper may be used in almost any modern functional programming language.

In section 2 we will see how a simple quadtree data type can be defined and used. We will build on this example to illustrate lazy evaluation and the sharing of common substructures in sections 3 and 4, respectively. In section 5 we will generalize the original quadtree definition to quadtrees that can consistently contain leaf values of any type. The advantages of higher order functions will be illustrated in section 6. In section 7 we will introduce the raster abstract data type and give algorithms for converting between rasters and quadtrees. (The complete raster to quadtree conversion algorithm is only 15 lines long, and could be reduced to 8 if we were more concerned with compactness than simplicity.) The coroutine behavior of functional programs is discussed in section 8, and modularity is considered in section 9. (The modularity features in section 9 are similar to those in some modern procedural languages. This section is included to show that these features can be supported in a functional language as well.) Parallel evaluation of functional programs is briefly considered in section 10. Section 11 is the conclusion.

2 Quadtrees

The Miranda type definition

```
quadtree ::= Black | White |
             Gray(quadtree, quadtree, quadtree, quadtree)
```

corresponds roughly to the following collection of Pascal type definitions.

```
type quadtree = ^node;
     color = (black, white, gray);
     node = record
                 case tag: color of
                      black, white: ();
                      gray: (nw, ne, sw, se: quadtree)
            end;
```

In Miranda, *Black*, *White* and *Gray* are constructor functions[2] for the type *quadtree* as well as "tags". No explicit use of pointers is required in a recursive type definition. It is not necessary to name the components of a data structure. For example, the value of the Miranda expression

```
Gray(Black, Gray(Black, Black, White, White), White, Black)
```

[1]Miranda is a trademark of Research Software Ltd.

[2]In Miranda, constructor function and only constructor functions start with upper case letters.

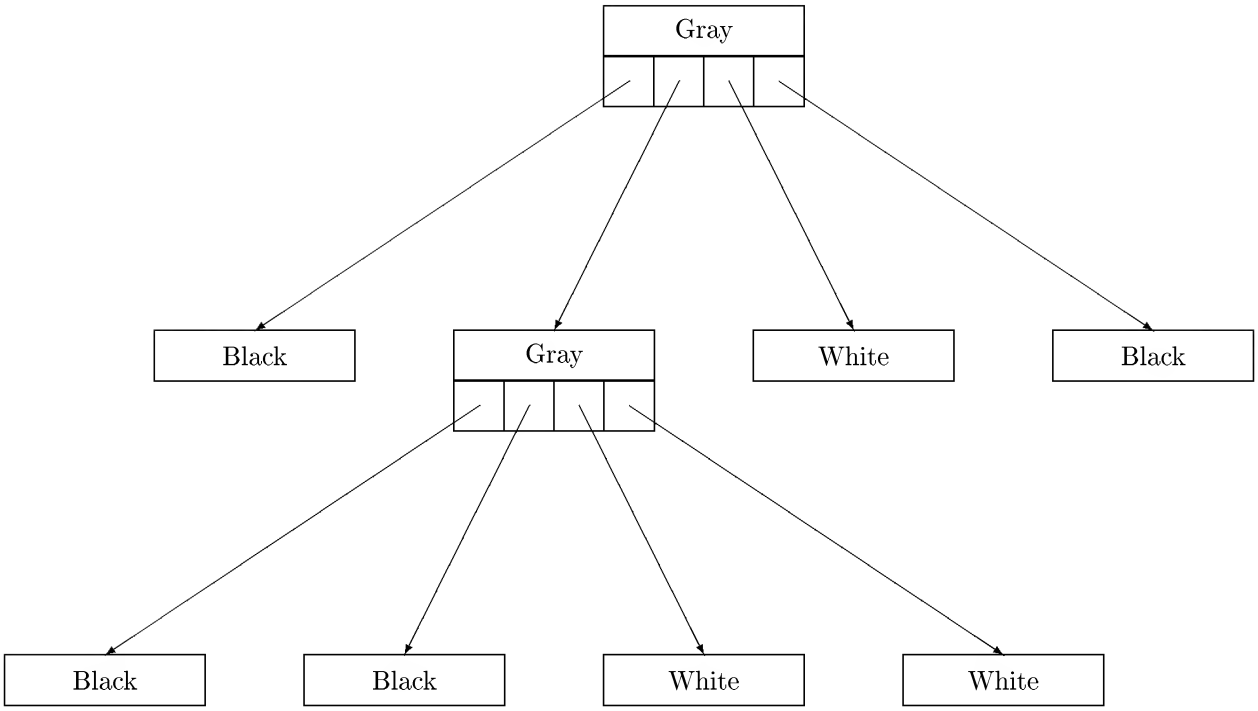

Figure 1: Quadtree representation of the image

is the quadtree of Fig. 1.

A raster representation of an image in a computer is an array of pixels (picture elements) where each pixel represents a single color or intensity. This same information can be represented more compactly by using a quadtree data structure, provided nearby pixels tend to have the same value. In a quadtree, if all pixels contain the same value, then the image is represented by a single leaf node containing that value. Otherwise the root node has four children. The image is quartered and each subtree of the quadtree represents one of the four subimages, in a recursive manner. Quadtrees, rather than binary trees, are used so that at each level the image can be halved in each dimension. We will use the term *node* to denote a node in a quadtree and the term *cell* to denote a square region in an image corresponding to a quadtree node (usually a quadtree leaf). For example, the black and white image represented by the 8×8 pixel array in Fig. 2 could be more compactly represented by the quadtree in Fig. 1, with the subtrees ordered: northwest, northeast, southwest, and southeast. The cell corresponding to the leaves of the quadtree of Fig. 1 are shown in Fig. 3. A good introduction to quadtrees may be found in [**?**].

A region (e.g. a geographic area) can be represented as a black and white image. We will regard black as denoting inside and white outside. A Miranda function to compute the intersection of the regions represented by two quadtrees is shown in Fig. 4[3]. The *intersection* function consists of a declaration of the type of the function followed by a collection of equations. Pattern matching is used to determine which equation to apply to a given pair of quadtrees. In patterns a data constructor function, which will start with an upper case letter, must match exactly. Any other identifier will denote a variable that will match anything. For example, if the first argument of an application of *intersection* has value *Black* then the second equation is

[3]The prime character is an ordinary character with no special meaning in a Miranda identifier.

Black	Black	Black	Black	Black	Black	Black	Black
Black	Black	Black	Black	Black	Black	Black	Black
Black	Black	Black	Black	White	White	White	White
Black	Black	Black	Black	White	White	White	White
White	White	White	White	Black	Black	Black	Black
White	White	White	White	Black	Black	Black	Black
White	White	White	White	Black	Black	Black	Black
White	White	White	White	Black	Black	Black	Black

Figure 2: Raster representation of an image

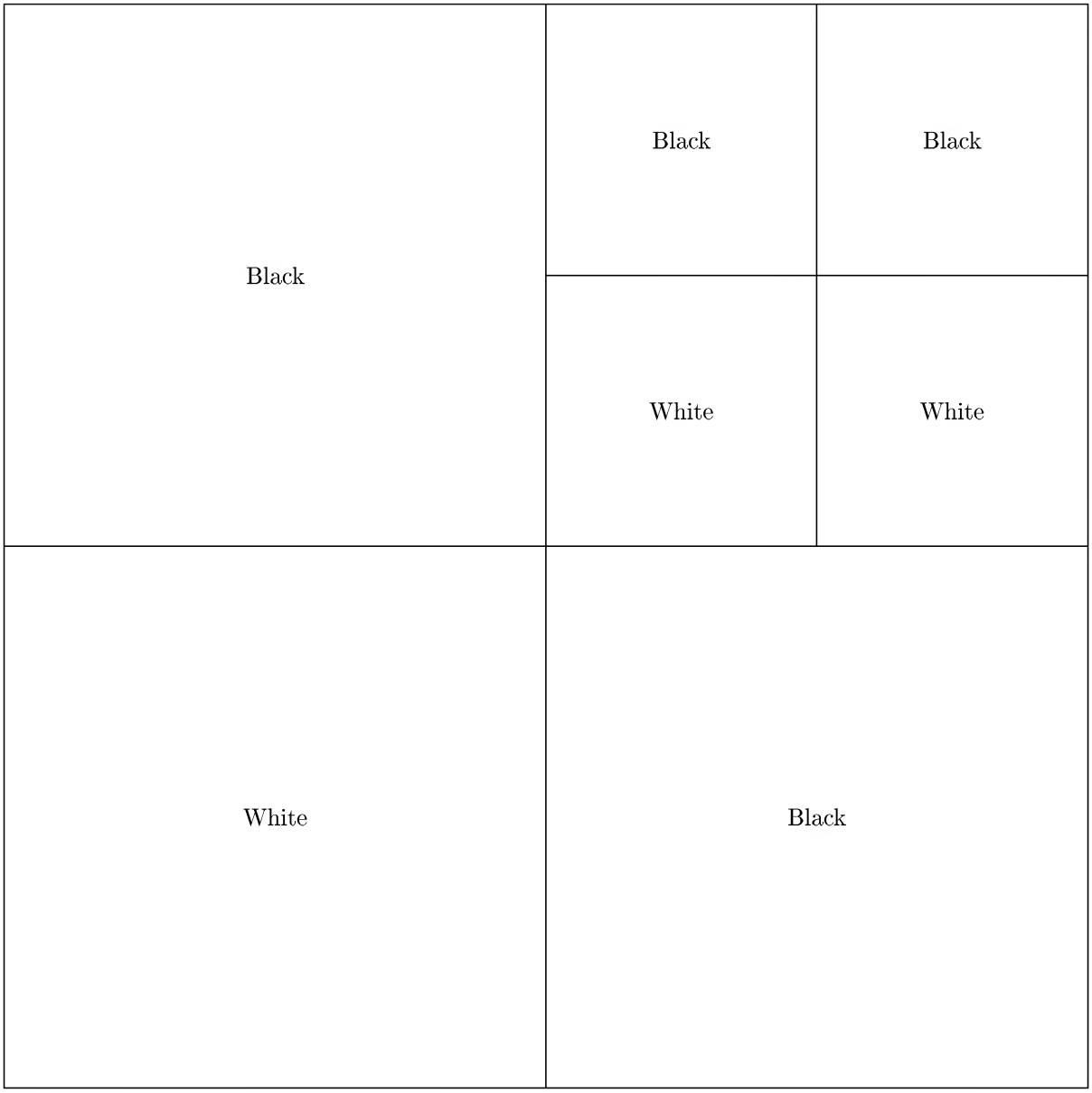

Figure 3: Location of cells in the quadtree

```
intersection :: (quadtree, quadtree) -> quadtree
intersection(White, any) = White
intersection(Black, any) = any
intersection(any, White) = White
intersection(any, Black) = any
intersection(Gray(nw, ne, sw, se), Gray(nw', ne', sw', se'))
   = Gray(intersection(nw, nw'),
          intersection(ne, ne'),
          intersection(sw, sw'),
          intersection(se, se'))
```

Figure 4: Simple *quadtree* intersection function

used and the result returned is the value of the second argument. Since objects can never change their values in a functional programming language, it is not necessary to copy the second argument (which could be a very large tree). Internally, the result is represented by a pointer to the existing tree. If pattern matching will work with more than one equation, then the first equation that can be used will be used. For example, when evaluating *intersection(Black,White)*, both the second and third equations match the pattern of the arguments, so the second equation will be used. If neither quadtree is entirely black or entirely white, then the last equation is used and the intersections of corresponding subtrees are computed and combined into a new gray subtree. Pattern matching can be efficiently compiled [**?**]. The run-time testing that is required is the same as would be required to test the value of discriminates in a language such as Pascal.

There is one minor problem with the *intersection* function as given. It is possible to produce gray nodes containing all white leaf nodes. For example,

```
result = intersection
            (Gray(White, Black, White, White),
             Gray(Black, White, White, Black))
```

will compute *Gray(White, White, White, White)* rather than *White* for *result*. The solution to this problem is to apply the function *fix* defined by:

```
fix :: (quadtree) -> quadtree
fix(Black) = Black
fix(White) = White
fix(Gray(nw, ne, sw, se)) =
   merge(Gray(fix(nw), fix(ne), fix(sw), fix(se)))
   where
      merge(Gray(Black, Black, Black, Black)) = Black
      merge(Gray(White, White, White, White)) = White
      merge(other) = other
```

to the result of an intersection operation. The *where* part of an equation is used to define variables and functions that are local to the one equation. The scope of the function *merge* is the last of the three equations defining *fix*. Now

```
result = fix(intersection(
                Gray(White, Black, White, White),
                Gray(Black, White, White, Black))
```

will compute *White* for *result*. Notice that *fix* combines leaves recursively from the bottom up, so that an arbitrary number of leaves may be combined into a single leaf when appropriate. It is tempting to incorporate *fix* into the *intersection* function. However, as we will see shortly, it is often very much more efficient not to fix intermediate results in a compound calculation.

3 Lazy Evaluation

Miranda uses lazy evaluation. Lazy evaluation is an optimization on call-by-name. As with call-by-name, no argument is evaluated until its value is needed. This includes arguments to data constructors. Once an expression has been evaluated, its value is saved, so no expression needs to be evaluated more than once. This is safe in a functional language, since the absence of side effects insures that an expression must yield the same value every time it is evaluated.

Lazy evaluation will often cause programs to be surprisingly efficient. Consider the problem of finding the intersection of the three regions in Fig. 5. If we were programming in a language like Pascal and had a pairwise intersection function, we might consider which pair to intersect first. However, the intersection of any pair of these regions may require an arbitrarily large amount of work, determined by the resolution of the image.

In Miranda, and other lazy functional languages, we do not need to worry about this problem. With lazy evaluation, intermediate results are computed only to the extent necessary to compute the overall result. An approximate trace [4] of the evaluation of this intersection can be found in Fig. 6. Each occurance of "?" denotes a different *Gray* quadtree. For each of these subtrees, we need never know more than that it is gray. At each step in the computation, Miranda selects an outermost expression that it can match to the left-hand side of an equation in Fig. 4 and replaces the expression by the corresponding right-hand side. When *fix*ed, the final result will simplify to *White*.

Lazy evaluation can be simulated in a procedural language, but it is awkward at best [**?**].

In section 8 we will see that lazy evaluation also can lead to storage economies. We should note that there are cases where lazy evaluation can cause significant problems with space efficiency. However, a compiler optimization [**?**] can eliminate most of these problems. The rest can be eliminated in a language supporting call-by-value and call-by-name [**?**].

[4] In practice, subtrees will be evaluated one at a time. To reduce the size of the figure, we show them being evaluated in parallel.

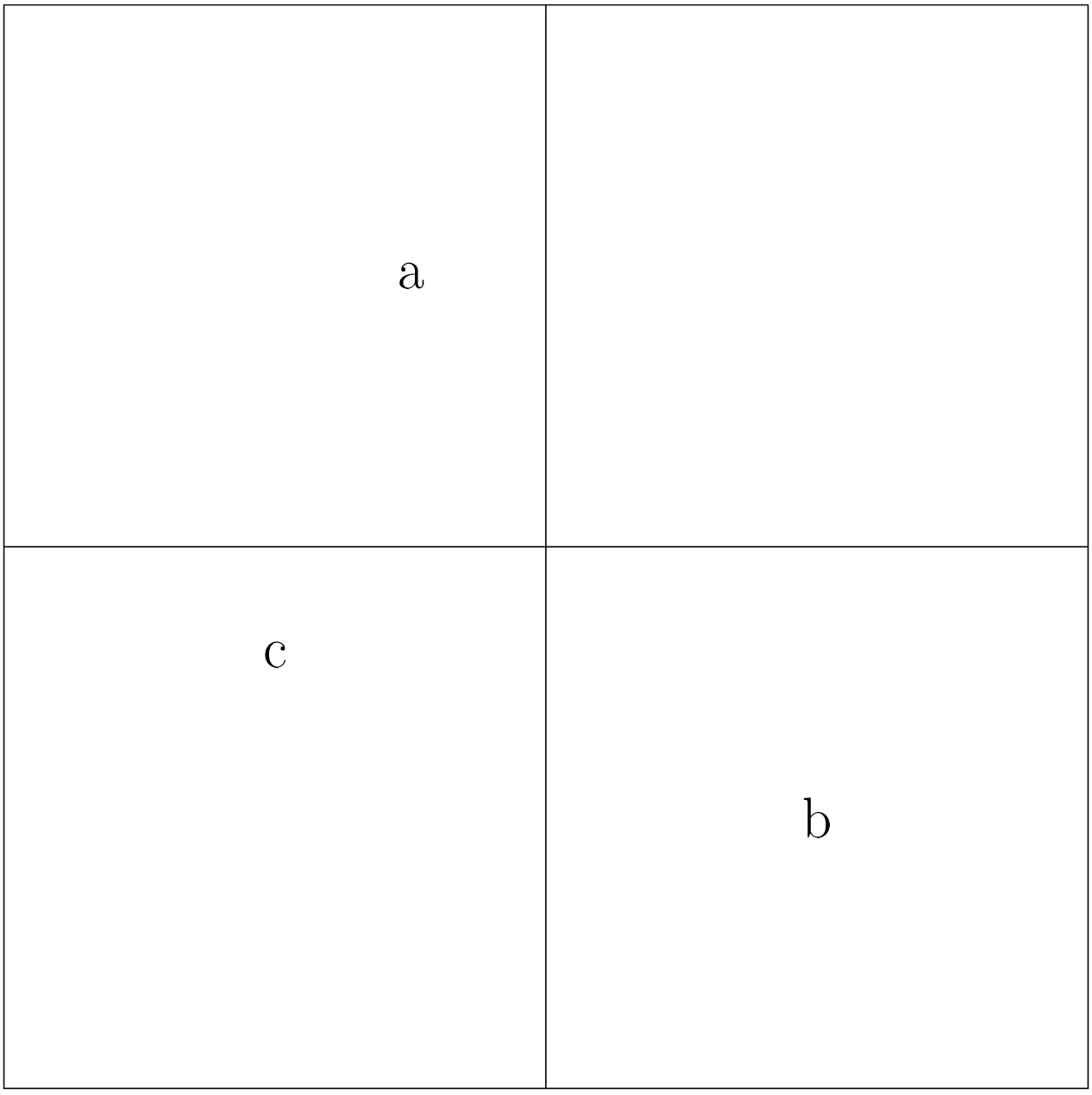

Figure 5: Three pairwise intersecting regions

```
intersection(a, intersection(b, c))

= intersection(
     Gray(?, ?, White, White),
     intersection(
        Gray(White, ?, ?, ?),
        Gray(?, White, ?, White)))

= intersection(
     Gray(?, ?, White, White),
     Gray(
       intersection(White, ?),
       intersection(?, White),
       intersection(?, ?),
       intersection(?, White)))

= Gray(
     intersection(?, intersection(White, ?)),
     intersection(?, intersection(?, White)),
     intersection(White, intersection(?, ?)),
     intersection(White, intersection(?, White)))

= Gray(
     intersection(?, White),
     intersection(?, White),
     White,
     White)

= Gray(White, White, White, White)
```

Figure 6: Approximate trace of the evaluation of *result 5*

4 Common Substructures

As mentioned earlier, objects in a functional program never change their values. (Instead we may have a number of different objects with the same name at different levels of recursion.) Since objects can not change, it does not matter whether a new object contains a pointer to, or a copy of, some other object, at least as far as the correctness of the implementation is concerned. This is why functional languages do not have explicit pointer types. Pointers are a low level concept of concern to the implementation only.

By sharing common substructures, it is possible to save both the time and the space that would otherwise be required to copy the substructures. For example, Fig. 7 shows the cells corresponding to the leaf nodes of two overlapping circles approximated by quadtrees. (Actually, three regions are shown. Near the intersection, the figure has more cells than would be required for either circle without the other. A cell is subdivided if it would have to be subdivided to represent either region 1 or region 2, even if it would not need to be subdivided for the other region.) The size (in nodes) of the quadtree required to represent either circle or the region of intersection grows approximately linearly with the resolution (or equivalently, exponentially with the depth of the quadtree). Since the size of the result of computing the intersection grows linearly with the resolution, one would expect the time required to compute the intersection to also grow linearly with the resolution. In fact, it grows logarithmically with the resolution provided both of the quadtrees representing the circles have already been generated in full. The reason for this is that the intersection will contain (pointers to) portions of both of the previously generated circle quadtrees. In general, the intersection algorithm needs to go to a greater depth of recursion if and only if the current image cell contains parts of the boundaries of both circles. That is, the intersection algorithm only goes full depth near the points of intersection, so requires time approximately proportional to the depth of the quadtree.

It is nice to be able to generate a result in time that grows only logarithmically with the size of the result. Of course, the amount of new storage that must be allocated is proportional to the time required to compute the result. It is useful to distinguish between the size of an object and its marginal size relative to other structures. For example, the size of the result of the intersection grows linearly with the resolution, but its marginal size relative to the two circles grows only logarithmically with the resolution. Of course, if one or both of the circles become available for garbage collection before the area of intersection does, then the shared storage cannot be reclaimed until the area of intersection is also available for garbage collection.

5 Polymorphism

Quadtrees are useful for other than black and white images. For example, if an area is divided into political districts, a quadtree may be used to represent the partition. A district number, or other district identifier, can be kept in each leaf node[5].

If one wants several types of quadtrees in a language like Pascal, there are two reasonable alternatives. One alternative is to define a quadtree of numbers and interpret the numbers differently in different quadtrees. (e.g. In a black and white quadtree, 0 might mean white, 1 black, and any other value would be an error.) The other alternative is to use a text editor to produce a different quadtree type declaration, with a slightly

[5]We have used quadtrees with a character string in each leaf to generate some of the figures in this paper.

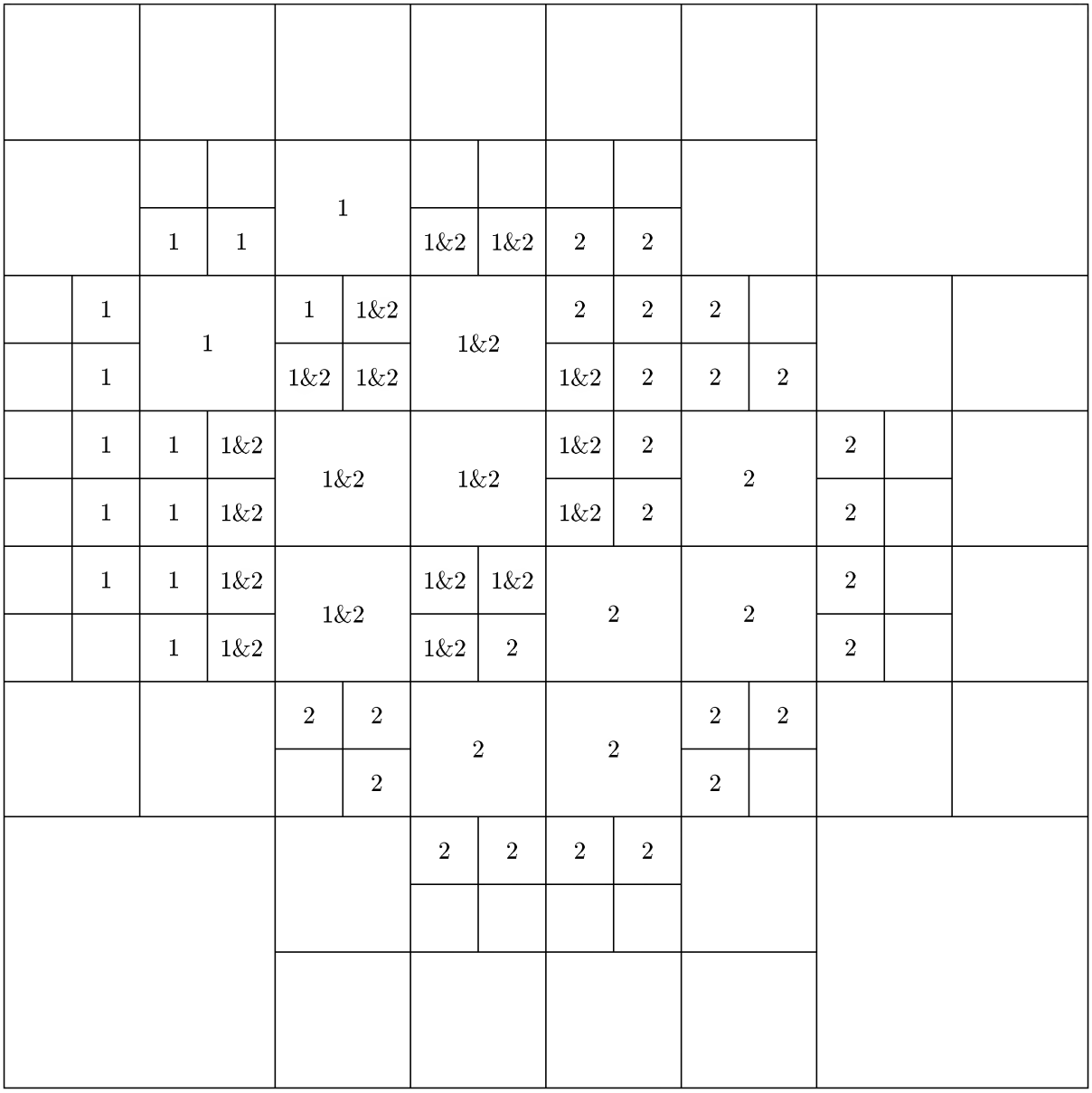

Figure 7: Quadtrees of two overlapping circles

different name, for each quadtree type. Ada generic packages make it possible to define a generic quadtree. However, for each quadtree type, a separate quadtree package instance must be created and used. Miranda supports polymorphic types and functions [?]. Strong type checking is possible with polymorphic types and functions. However, unlike Ada, it is not necessary to generate specific instances of polymorphic types and functions.

We can define a polymorphic quadtree type and a new *fix* function by:

```
qtree * ::= Leaf(*) | Internal(qtree(*), qtree(*), qtree(*), qtree(*))


fix :: qtree(*) -> qtree(*)
fix(Leaf(x)) = Leaf(x)
fix(Internal(nw, ne, sw, se)) =
   merge(Internal(fix(nw), fix(ne), fix(sw), fix(se)))
   where
      merge(Internal(Leaf(x), Leaf(x), Leaf(x), Leaf(x)))
         = Leaf(x)
      merge(other) = other
```

Notice that the variable x appears four times in the pattern in the first equation defining *merge*. The pattern matches if and only if all occurrences of x match the same value. To avoid confusion, we have chosen to call the polymorphic form *qtree*[6]. The type variable, "*", is a variable that can take any type as a value. For any type *t*, *qtree(t)* is a type. For example, we could represent black and white quadtrees with quadtrees of boolean values, of type *qtree(bool)*. Alternatively, we could define an enumeration type

```
color ::= Black | White
```

and use type *qtree(color)*. These types are isomorphic to the earlier type *quadtree*.

Even though every object in Miranda has a type associated with it, declaring the type of an object is optional. Miranda can determine a type if it is not declared. For example, in

```
a = Internal(Leaf(True), Leaf(False), Leaf(True), Leaf(True))
b = Internal(Leaf("True"), Leaf("x"), Leaf("y"), Leaf("z"))
c = Internal(Leaf("x"), Leaf(False), Leaf(True), Leaf(True))
```

object *a* is of type *qtree(bool)*, *b* is of type *qtree([char])*, and *c* causes a type error since one of its leaves contain a character string and the others contain booleans. The type *[char]* is list of *char*.

Some functions, such as *fix*, can process *qtree*s of any type, while others, such as *intersection* (which will be reconsidered shortly), are restricted to more specific *qtree* types.

In practice, a language with polymorphic types allows one almost as much freedom as an untyped language such as Lisp, while detecting all type errors at compile time.

[6]We will assume that type *qtree* and type *quadtree* are never in scope at the same time. Otherwise there will be various name conflicts (e.g. *fix*, etc.)

6 Higher Order Functions

A function that takes a function as a parameter or returns a function as a result is called a higher order function. Higher order functions are useful for abstracting out common computational processes, and make programs more modular. Some higher order functions correspond to control structures in procedural languages. Miranda and other similar languages provide a collection of standard higher order functions in much the same way that conventional languages provide standard control structures. However, in a functional language it is possible to define additional higher order functions. It is often useful to define new higher order functions to process programmer defined data structures. By analogy, the for-loop structure is useful for processing arrays in Pascal, but the programmer has no way of defining new control structures to iterate over programmer defined tree structures.

Fig. 8 lists several general purpose functions for processing *qtree*s. We will consider each in turn.

The function *apply_to_leaves* is a functional counterpart to a control structure for iterating over the leaves of a *qtree*. The result of applying *apply_to_leaves(h)* to q will be q with each leaf value x replaced by *h(x)*. For example, the function *apply_to_leaves(∼)* will compute the complement of a *qtree(bool)*, where "∼" is the logical negation operator. That is, the value of *apply_to_leaves(∼)* is the complement function for *qtree(bool)*. As another example, we have a function *draw_tree*, which produces LaTeX commands to draw figures. It only works on quadtrees of type *qtree([char])*. We tend to use objects of type *qtree(bool)* in various computations. To draw a quadtree q with labels "Black" for *True* and "White" for *False* we can define

```
bool_to_chars :: (bool) -> [char]
bool_to_chars(True) = "Black"
bool_to_chars(False) = "White"

output = draw_tree(transform(q))
         where
            transform = apply_to_leaves(bool_to_char)
```

and evaluate *output*.

The function *apply_to_subtrees* is similar to *apply_to_leaves*, but applies a function to the children of an internal node rather than to leaves. It is defined only for internal nodes. The function *apply_to_leaves* is used in the definition of the function *limit_depth*. We will examine these two functions together. The function *limit_depth* is used to limit the depth of a *qtree*, that is, to compute a new subtree similar to the argument but of bounded depth. If the given subtree is too deep, then appropriate subtrees are replaced by leaves. We arbitrarily pick the value at the upper left hand corner of the subtree image as the new leaf value. Notice how *apply_to_subtrees* is used in the final case of *limit_depth* to recursively apply *limit_depth* to the subtrees.

There are two primary uses for *limit_depth*. First, it may be used to limit the depth of a quadtree that is to be displayed on a device having fixed resolution, or is to be converted to another representation (see section 7). The other use is to control the resolution used in a computation. For example, suppose we want to compute the area of that region in a national park that has more than 20 inches of rain per year. We may well have the park boundary defined to a high resolution, but a quadtree representation of a rainfall contour map would probably have a much lower resolution. If we intersect the park and the region with 20

```
apply_to_leaves :: (* -> **) -> (qtree * -> qtree **)
apply_to_leaves(f) =
   g
   where
      g(Leaf(x)) = Leaf(f(x))
      g(Internal(nw, ne, sw, se)) =
         Internal(g(nw), g(ne), g(sw), g(se))

apply_to_subtrees ::
   (qtree * -> qtree **) -> (qtree * -> qtree **)
apply_to_subtrees(f) =
   g
   where
      g(Internal(nw, ne, sw, se)) =
         Internal(f(nw), f(ne), f(sw), f(se))

limit_depth :: (num) -> (qtree * -> qtree *)
limit_depth(0) =
   pick
   where
      pick(Leaf(x)) = Leaf(x)
      pick(Internal(nw, ne, sw, se)) = pick(nw)
limit_depth(n) =
   f
   where
      f(Leaf(x)) = Leaf(x)
      f(other) = (apply_to_subtrees(limit_depth(n-1))) (other)

apply_to_subtrees2 ::
   ((qtree *, qtree **) -> qtree ***) ->
   ((qtree *, qtree **) -> qtree ***)
apply_to_subtrees2(f) =
   g
   where
      g(Internal(nw, ne, sw, se), Internal(nw', ne', sw', se')) =
         Internal(f(nw, nw'), f(ne, ne'), f(sw, sw'), f(se, se'))

expand :: qtree(*) -> qtree(*)
expand(Internal(nw, ne, sw, se)) = Internal(nw, ne, sw, se)
expand(Leaf(x)) = Internal(y, y, y, y)
                  where
                     y = Leaf(x)
```

inches of rain, then we will get a high resolution quadtree representing a region that is much less precisely defined. By using *limit_depth*, we can produce a quadtree with an appropriate resolution instead.

The reader used to programming in a procedural language is probably thinking that this is a rather inefficient solution, to compute a high resolution intersection and then throw much of it away by applying *limit_depth*. However, with lazy-evaluation, only that part of the intersection quadtree that is actually accessed by *limit_depth* is ever generated. Lazy evaluation makes it possible to write more modular programs. Consider the computation of

```
(limit_depth(k))(intersection(a, intersection(b, c)))
```

using lazy evaluation. In addition to the advantages of lazy evaluation in computing the nested intersections, as considered in section 3, we have the advantage of having to specify the resolution of the overall computation only once. In a procedural language, resolution information usually would be passed to each *intersection* application to avoid wasted work.

The *apply_to_subtrees2* function applies a function to corresponding subtrees of two quadtrees. For example,

```
intersection :: (qtree(bool), qtree(bool)) -> qtree(bool)
intersection(Leaf(False), any) = Leaf(False)
intersection(Leaf(True), any) = any
intersection(any, Leaf(False)) = Leaf(False)
intersection(any, Leaf(True)) = any
intersection(x, y) = (apply_to_subtrees2(intersection)) (x, y)
```

defines a *qtree intersection* function. The reader may wish to compare this with the *quadtree intersection* function in Fig. 4.

The map overlay problem is a generalization of the problem of computing intersections. Given two maps, the problem is to produce a third map where the value at each point is a function of the values of the arguments at the corresponding points. The function

```
overlay :: (*, **) -> ***) ->
           ((qtree(*), qtree(**)) -> qtree(***))
overlay(f) =
   g
   where
   g(Leaf(a), Leaf(b)) = Leaf(f(a, b))
   g(x, y) =
      (apply_to_subtrees2(overlay(f))) (expand(x), expand(y))
```

does this. Fig. 7 was generated by

```
result = draw_tree((overlay(which)) (circle1, circle2))


which(False, False) = ""
```

```
which(True, False) = "1"
which(False, True) = "2"
which(True, True) = "1&2"
```

where *circle1* and *circle2* are of type *qtree(bool)*. It is possible to overlay quadtrees of different types using *overlay*, and the result may be a quadtree of a third type. The programmer is allowed to provide an arbitrary function for combining leaf values of the two quadtrees to produce the leaf values for the result. Notice that a cost of the increased generality of *overlay* is that recursion does not terminate until leaves are reached in both trees, while *intersection* can terminate when either tree is a leaf.

Finally, the function *expand*, while not a higher order function, is generally useful, and so in included in the functions listed in Fig. 8. It was used in the definition of *overlay*, for example.

7 Rasters

The raster representation of an image is a array of pixels. Each pixel contains the same type of information as is found in quadtree leaves. We will restrict our attention to square arrays, of size $2^k \times 2^k$ for some integer k.

Some functional languages support arrays, but Miranda does not. Most of the algorithms that use arrays in procedural languages can be simply written using lists in a functional language. Miranda has a built in polymorphic list data type. The empty list is denoted by *[]*. The colon is an infix cons operator. That is, if a is an element and x is a list, then $a : x$ is a new list produced by adding a to the front of the list x. If t is any type, then *[t]* is the type of a list having elements of type t. For example, *1 : 2 : 3 : []* is the list of type *[num]* containing 1, 2 and 3.

We will use a list of lists to represent a raster. Each row of the raster will be a list of pixels ordered from left to right, and the raster will be a list of rows ordered from top to bottom. Pixels can be of any type. The polymorphic type *raster* can be defined by

```
raster * == [[*]].
```

Fig. 9 contains an algorithm to convert a raster to a quadtree. Recall that pixels can be of any type. The conversion algorithm uses a hybrid representation consisting of a raster of quadtrees for intermediate results. Each intermediate result denotes the same image.

The conversion algorithm starts by computing a raster of quadtrees where each quadtree is a single leaf. The expression

```
[Leaf pixel | pixel <- row]
```

means "the list of all elements *Leaf pixel* where *pixel* is in the list *row*". This is analogous to the mathematical set expression

$$\{Leaf\ \ pixel \mid pixel \in row\}.$$

If we apply the function *join_rows* to a raster of quadtrees it will return another raster of quadtrees, denoting the same image, with only a quarter as many quadtrees, but with each quadtree having depth

```
raster_to_qtree :: raster(*) -> qtree(*)
raster_to_qtree = fix.raster_to_full_tree

raster_to_full_tree(array) =
   combine([[Leaf pixel | pixel <- row] | row <- array])

combine :: ([[qtree(*)]]) -> qtree(*)
combine([[root]]) = root
combine(other) = combine(join_rows(other))

join_rows :: ([[qtree(*)]]) -> [[qtree(*)]]
join_rows([]) = []
join_rows(first:second:rest)
   = join_pair(first, second) : join_rows(rest)

join_pair :: ([qtree(*)], [qtree(*)]) -> [qtree(*)]
join_pair([], []) = []
join_pair(nw:ne:rest, sw:se:rest') =
   Internal(nw, ne, sw, se) : join_pair(rest, rest')
```

Figure 9: Raster to quadtree conversion algorithm

one greater than before. Groups of four quadtrees are joined together. A pair of rows is combined by each application of *join_pair*. The function *combine* calls *join_rows* repeatedly, until a 1×1 raster results. Since each hybrid raster of quadtrees structure denotes the same image, the desired result is the quadtree in the single pixel of the raster.

We get the desired *raster_to_qtree* function by composing the two functions *fix* and *raster_to_full_tree*. The period is an infix function composition operation.

8 Coroutines

Most of the other published raster to quadtree and quadtree to raster conversion programs are considerably more complicated than the above program. The reason is that it is usually undesirable to have the entire raster representation in memory at any one time. (In general, quadtree storage requirements tend to vary linearly with the resolution while raster storage requirements vary quadratically with the resolution.) In effect, procedural programs must simulate a coroutine structure, so that each row of a raster can be incorporated into a quadtree before the next row is generated.

Recall that with lazy evaluation, no expression is evaluated until it is required. Similarly, input is not read until it is needed. Lazy evaluation gives us the desired behavior for free. The above raster to quadtree algorithm is optimal (to within a constant factor) in both time and space. The advantage of functional programming here is that the order in which things are computed can be very different from the order in which they are defined in a functional program. Conceptually, in the raster to quadtree algorithm, we start with a full raster and repeatedly transform it into smaller rasters of deeper quadtrees. In fact, with lazy evaluation, much of the final quad tree can be in existence (and the portion of the raster that generated it may have been reclaimed by the storage management system) before the last of the raster has been generated.

9 Modularity

Modularity is an important concept in any good modern general purpose programming language.

As we have seen, functional programming leads to increased modularity in two important ways. With lazy evaluation, the order in which we compute objects does not need to relate to the order in which we define the objects. For example, conceptually we can define a very high resolution quadtree and then apply *fix* to give us the resolution that we require. In practice, the high resolution quadtree is never fully generated. The *fix* function and the high resolution quadtree expression interact in a coroutine like manner. This permits us to define functions in a conceptually modular fashion, even when efficiency requires interleaving the execution of the various functions.

Higher order functions are another example of modularity. A functional programming language allows us to abstract common patterns of behavior and encapsulate them in higher order functions.

In recent years, we have learned a great deal about the value of abstract data types, and modules with import and export controls, for programming in the large. It is important that any serious modern programming language include such features in some reasonable form. Miranda, and various other functional languages, do.

```
>|| Module raster
%export raster  break_raster  compute_raster  raster_to_qtree
        qtree_to_raster  raster_to_full_tree
%include "general"
%include "frame_of_reference"

abstype raster *
with  break_raster :: (raster(*)) -> [[*]]
      compute_raster :: (num, (num, num) -> *) -> raster(*)
      raster_to_full_tree :: (raster(*)) -> qtree(*)
      qtree_to_raster :: (num, qtree(*)) -> raster(*)

raster * == [[*]]

break_raster array = array

compute_raster(k, f) =
   [[ f(xmin + width * row, ymin + width - width * col)
      | row <- steps] | col <- steps]
   where
      steps = [half, 3 * half..1.0]
      half = 0.5/(2^k)
```

Implementation of `raster_to_qtree`, `qtree_to_raster` *and* `raster_to_full_tree`.

Figure 10: The module defining the type *raster* and related operations

Fig. 10 includes a definition of *raster* as an abstract data type. While we will not be concerned about details here, we will note that the abstract data type definition permits only the four functions *break_raster*, *compute_raster*, *raster_to_full_tree* and *qtree_to_raster* to access the representation of a raster. Since the function *raster_to_quadtree* is defined in terms of the *raster_to_full_tree* function, it does not need to access the representation.

It is important that type *raster* be an abstract data type, to insure that no ill-formed rasters are ever generated. (Note that the *raster_to_full_tree* algorithm will fail unless the length of each row is equal to the number of rows and the number of rows is a power of two.)

The export control permits us to keep names such as *join_rows* local to the module, so that name spaces do not become too cluttered. If desired, we could put additional information in the *include* statements and import only selected components of other modules.

10 Parallelism

One of the major advantages of functional programming languages is that functional programs usually have a high potential for parallelism. Since the evaluation of an expression cannot have any side effects, such as changing the value of a global variable, independent subexpressions may be evaluated in any order, or in parallel. For example, if the subtrees of a quadtree to which *fix* is applied are *fix*ed in parallel, then a considerable amount of parallelism can result.

There are also other forms of parallelism possible. For example, a raster is represented as a list of lists. If the top level list is buffered, so the next row of an image is generated while the current row is being output, additional parallelism is possible.

There is a considerable amount of research currently being done on the parallel evaluation of functional programs[**?**, **?**, **?**]. With parallelism being the most promising way to get faster computational speeds at an affordable price, it is likely that functional programming languages will grow in popularity.

11 Conclusion

As we have seen, a functional program often is both simpler and more efficient than the obvious procedural program to solve a problem.

One reason why functional programs are simpler is that the concise mathematical notation gives a static declaration of the required result. The programmer does not need to think in terms of a constantly changing state. This is perhaps most helpful in programming in the small.

In addition, modern functional programming languages support modularity in a number of ways, making programming in the large much simpler. Miranda supports modules (or packages) and abstract data types in much the same way that modern procedural languages do. In addition, the coroutine behavior of lazy evaluation makes it possible to package processes as simple functions and still get the efficiency that results from applying these processes piecewise, as and when required. Finally, polymorphic higher order functions can encapsulate common patterns of behavior in a very general manner.

Time efficiency results both from using lazy evaluation to avoid generating unneeded components of intermediate results and from the sharing of common substructures that might otherwise need to be copied. Of course common substructures can be shared in procedural languages, but the programmer must then know whether a substructure that may be modified is shared or copied, and take appropriate action when each modification is made. These issues cease to exist in a functional language, where object can never change.

Space efficiency results from the sharing of common substructures and the coroutine behavior that lazy evaluation produces.

Functional programming languages may be used either for production programming or for formal specification and rapid prototyping.

Acknowledgement: The authors would like to thank the referees, and Adam Beguelin and Carolyn Schnauble, of the University of Colorado, for their many helpful suggestions.